游荡在思考的迷宫中

甄景贤 (King-Yin Yan) and Juan Carlos Kuri Pinto

General.Intelligence@Gmail.com

Abstract. 简单介绍笔者的强人工智能理论和强化学习、动态规划、最优控制、Hamiltonian 力学系统的关系。

1 经典逻辑 AI 背景

Strong AI 的问题在理论上已经被数理逻辑完整地描述了，馀下的问题是学习算法，因为在逻辑 AI 的架构下，学习算法很慢（复杂性很高），这就是我们要解决的。

我研究 logic-based AI 很多年，因此我的思路喜欢将新问题还原到逻辑 AI 那边去理解，但实际上我提倡的解决办法不是靠经典逻辑，甚至不是 symbolic 的。但在这篇文章我还是会经常跳回到逻辑 AI 去方便理解。

用数理逻辑模拟人的思想是可行的，例如有 deduction, abduction, induction 等这些模式，详细可见《Computational logic and human thinking》by Robert Kowalski, 2011. 这些方面不影响本文的阅读。值得一提的是，作者 Kowalski 是 logic programming，特别是 Prolog，的理论奠基人之一。

在经典逻辑 AI 中，「思考」是透过一些类似以下的步骤：

$$\text{前提} \vdash \text{结论} \tag{1}$$

$$\boxed{\text{今天早上下雨}} \vdash \boxed{\text{草地是湿的}} \tag{2}$$

亦即由一些命题(propositions) 推导到另一些命题。

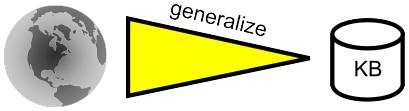
推导必须依靠一些逻辑的法则命题 (rule propositions)，所谓「法则」是指命题里面带有 x 这样的变量(variables)：

$$\boxed{\text{地方 x 下雨}} \wedge \boxed{\text{x 是露天的}} \vdash \boxed{\text{地方 x 是湿的}} \tag{3}$$

这些法则好比「逻辑引擎」的燃料，没有燃料引擎是不能推动的。

注意：命题里面的 x，好比是有「洞」的命题，它可以透过 substitution 代入一些实物 (objects)，而变成完整的命题。这种「句子内部」(sub-propositional) 的结构可以用 predicate logic （谓词逻辑）表达，但暂时不需要理会这些细节。

Logic-based AI 可以看成是将世界的「模型」压缩成一个「知识库」(knowledge-base, KB)，里面装著大量逻辑式子：

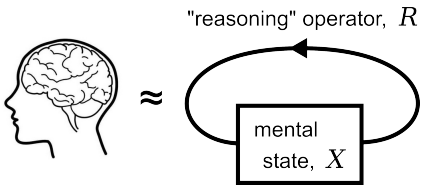


(4)

世界模型是由大量的逻辑式子经过组合而生成的，有点像向量空间是由其「基底」生成；但这生成过程在逻辑中特别复杂，所以符号逻辑具有很高的压缩比，但要学习一套逻辑知识库，则相应地也有极高的复杂度。

2 中心思想

关键是将「思考」看成是一个动态系统 (dynamical system)，它运行在思维状态 (mental states) 的空间中：



(5)

举例来说，一个思维状态可以是以下的一束命题：

- 我在我的房间内，正在写一篇 AGI-16 的论文。
- 我正在写一句句子的开头：「举例来说，....」
- 我将会写一个 NP (noun phrase)：「一个思维状态....」

思考的过程就是从一个思维状态 过渡 (transition) 到另一个思维状态。就算我现在说话，我的脑子也是靠思维状态记住我说话说到句子结构的哪部分，所以我才能组织句子的语法。

思维状态是一支向量 $\boldsymbol{x} \in X$，X 是全体思维空间，思考算子 (reasoning operator) R 是一个 automorphism 映射：$X \to X$。

换句话说：我们将逻辑 AI 的整套器材搬到向量空间中去处理。这个做法，部分是受到 Google 的 PageRank 和 Word2Vec [2] 算法的启发，因为它们都是在向量空间中运作，而且非常成功。

3 控制论

以下内容可以在一般「现代控制论」教科书中找到，例如：

- Daniel Liberzon 2012: *Calculus of variations and optimal control theory – a concise introduction*
- 李国勇 2008：《最优控制理论与应用》
- 张洪钺、王青 2005：《最优控制理论与应用》

一个**动态系统 (dynamical system)** 可以用以下方法定义：

$$\text{离散时间：} \quad \boldsymbol{x}_{t+1} = \boldsymbol{F}(\boldsymbol{x}_t) \tag{6}$$

$$\text{连续时间：} \quad \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) \tag{7}$$

其中 f 也可以随时间改变。如果 f 不依赖时间，则系统是 time-invariant （定常的），形式上如 (7) 那种微分方程叫作 autonomous （自主的）。

在我的智能系统理论里，我把 F 或 f 设定成 RNN (recurrent neural network)，即反馈式神经网络：

$$\text{离散时间：} \quad \boldsymbol{x}_{t+1} = \boxed{\text{RNN}}(\boldsymbol{x}_t) \tag{8}$$

$$\text{连续时间：} \quad \dot{\boldsymbol{x}} = \boxed{\text{RNN}}(\boldsymbol{x}) \tag{9}$$

这里 recurrent 指的是它不断重复作用在 $\boldsymbol{x}$ 之上，但实际上它是一个普通的前馈式 (feed-forward) 神经网络。注意：在抽象理论中，f 和 F 可以是任意函数，我把它们设计成 NN 只是众多可能的想法之一。之所以选用 NN，是因为它有 universal function approximator 的功能，而且是我们所知的最「聪明」的学习机器之一。

在我提出的智能系统里，$\dot{x}$ 是由學習機器給出的，换句話說，$\dot{x}$ 是思維狀態在梯度下降至最佳狀態時的方向導數。

一个（连续时间的）**控制系统 (control system)** 定义为：

$$\dot{\boldsymbol{x}(t)} = f(\boldsymbol{x}(t), \boldsymbol{u}(t), t) \tag{10}$$

其中 $\boldsymbol{u}(t)$ 是控制向量。控制论的目的就是找出最好的 $\boldsymbol{u}(t)$ 函数，令系统由初始状态 $\boldsymbol{x}_0$ 去到终点状态 $\boldsymbol{x}_\perp$。

注意：人工智能中的 **A* search**，是动态规划的一个特例。换句话说，用动态规划在某个空间中「漫游」，可以模拟到 best-first 搜寻的功能。

在这框架下，智能系统的运作可以分开成两方面：思考 和 学习。

思考即是根据已学得的知识（知识储存在 RNN 里），在思维空间中找寻 $\boldsymbol{x}$ 最优的轨迹，方法是用控制论计算 $\boldsymbol{u}^*$。$\boldsymbol{x}$ 的轨迹受 RNN 约束（系统只能依据「正确」的知识去思考），但思考时 RNN 是不变的。

学习就是学习神经网络 RNN 的 weights W。此时令 $u = 0$，即忽略控制论方面。

以上两者是两个独立的方面，但不排除它们可以在实际中同时进行。

3.1 控制论与强化学习的关系

在强化学习中，我们关注两个数量：

- $R(\boldsymbol{x}, a)$ = 在状态 $\boldsymbol{x}$ 做动作 a 所获得的奖励(reward)
- $U(\boldsymbol{x})$ = 状态 $\boldsymbol{x}$ 的效用(utility) 或 价值 (value)

简单来说，「价值」就是每个瞬时「奖励」对时间的积分：

$$\boxed{\text{价值 U}} = \int \boxed{\text{奖励 R}}\, dt \tag{11}$$

（价值有时用 V 表示，但为避免和势能 V 混淆故不用。）

用控制论的术语，通常定义 cost functional：

$$J = \int L dt + \Phi(\boldsymbol{x}_\perp) \tag{12}$$

其中 L 是 "running cost"，即行走每一步的「价钱」；Φ 是 terminal cost，即到达终点 $\boldsymbol{x}_\perp$ 时，那位置的价值。

在分析力学里 L 又叫 Lagrangian，而 L 对时间的积分叫「作用量」：

$$\boxed{\text{作用量 (Action)}}\ S = \int L dt \tag{13}$$

Hamilton 的最小作用量原理 (principle of least action) 说，在自然界的运动轨迹里，S 的值总是取稳定值 (stationary value)，即比起邻近的轨迹它的 S 值最小。

所以有这些对应：

强化学习	最优控制	分析力学
效用/价值 U	价钱 J	作用量 S
即时奖励 R	running cost	Lagrangian L
action a	control u	（外力?）

用比较浅显的例子：和美女做爱能带来即时的快感 (= 奖励 R)，但如果强奸的话会坐牢，之后很长时间很苦闷，所以这个做法的长远价值 U 比其他做法较低，正常人不会选择它。

有趣的是，奖励 R 对应於力学上的 Lagrangian，其物理学单位是「能量」；换句话说，「快感」或「开心」似乎可以用「能量」的单位来量度，这和通俗心理学里常说的「正能量」不谋而合。而，长远的价值，是以 [能量 × 时间] 的单位来量度。

一个智能系统，它有「智慧」的条件，就是每时每刻都不断追求「开心能量」或奖励 R 的最大值，但它必需权衡轻重，有计划地找到长远的效用 U 的最大值。

3.2 经典分析力学 (analytical mechanics)

分析力学的物理内容，完全是牛顿力学的 $F = ma$，但在表述上引入了能量和 Hamiltonian 等概念，再使用微积分和变分法。

Lagrange 方程

Lagrange 引入了 Lagrangian $L = T - V$，可以分拆成動能 T 和勢能 V 兩部分。

重點是：動能 T 是速度 $\dot{\boldsymbol{x}}$ 的函數，勢能 V 是位置 $\boldsymbol{x}$ 的函數。

问题：如果在强化学习中的「快感 / 奖励」对应於 Lagrangian L，如何在奖励之中分拆出「动能」和「势能」的分量？

$$\boxed{\text{Lagrange equation}} \quad \frac{d}{dt}\frac{\partial L}{\partial \dot{x}_i} - \frac{\partial L}{\partial x_i} = 0 \tag{14}$$

这些方程的座标是 $(\boldsymbol{x}, \dot{\boldsymbol{x}})$，可以了解成位置空间 (configuration space) 上的 tangent bundle （下述）。

Hamilton 方程

Hamiltonian $H = T + V$，亦即总能量，但它表示成位置 $\boldsymbol{x}$ 和动量 $\boldsymbol{p}$ 的函数。

$$\boxed{\text{Hamilton equation}} \quad \begin{cases} \dot{\boldsymbol{x}} &= \dfrac{\partial H}{\partial p} \\ \dot{\boldsymbol{p}} &= -\dfrac{\partial H}{\partial x} \end{cases} \tag{15}$$

这些方程的座标是**相位空间 (phase space)** $(\boldsymbol{x}, \boldsymbol{p})$。

位置空间和相位空间之间的变换是 **Legendre transformation**：

$$\boxed{\text{tangent bundle}}\ TX \to T^*X\ \boxed{\text{cotangent bundle}} \tag{16}$$

$$(\boldsymbol{x}, \dot{\boldsymbol{x}}) \mapsto (\boldsymbol{x}, \boldsymbol{p}) \tag{17}$$

$$\boldsymbol{p} = \frac{\partial L}{\partial \dot{\boldsymbol{q}}} \quad \Rightarrow \quad H := \boldsymbol{p}\dot{\boldsymbol{x}} - L \tag{18}$$

Hamilton-Jacobi 方程

$$\boxed{\text{Hamilton-Jacobi equation}} \quad H(q, \frac{\partial S}{\partial q}, t) + \frac{\partial S}{\partial t} = 0 \tag{19}$$

其中 S 是「作用量」。下面我们会用动态规划的原理推导出此一方程。

Poisson 括号

$$\boxed{\text{Poisson 括号}} \quad \{F, H\} := \sum_i \{\frac{\partial F}{\partial q_i}\frac{\partial H}{\partial p_i} - \frac{\partial F}{\partial p_i}\frac{\partial H}{\partial q_i}\} \tag{20}$$

在力学系统中，它表示任意一力学量（函数 f）对时间的改变量：

$$\dot{f} = \{f, H\} \tag{21}$$

$$\frac{\partial}{\partial t} = \frac{\partial}{\partial p}\dot{p} + \frac{\partial}{\partial q}\dot{q} \tag{22}$$

以上使用了微分的 chain rule，但由於 $\dot{\boldsymbol{p}}$ 和 $\dot{\boldsymbol{q}}$ 可以由 Hamilton 方程 (15) 给出，所以得到 Poisson 括号的形式 (20)。

每个物理学生都知道的「经典－量子对应原理」：

$$[\boldsymbol{F}, \boldsymbol{G}] \quad \Leftrightarrow \quad i\hbar\{F, G\} \tag{23}$$

这对应原理是 P.A.M. Dirac 发现的。

在经典力学里，

$$\{\boldsymbol{x}, \boldsymbol{p}\} = 1 \tag{24}$$

但在量子力学里，

$$[\boldsymbol{X}, \boldsymbol{P}] = i\hbar \tag{25}$$

这也是 Heisenberg 测不准原理的由来：

$$\Delta\boldsymbol{X}\Delta\boldsymbol{P} \geq \frac{\hbar}{2} \tag{26}$$

如果在某一流形上，「广义」Poisson 括号是 nondegenerate （非退化）的，则它变成了辛流形结构的 $\omega = \{\cdot, \cdot\}$ 括号（下述）。

3.3 Hamiltonian 的出现

考虑一个典型的控制论问题，系统是：

$$\text{状态方程：} \quad \dot{\boldsymbol{x}}(t) = \boldsymbol{f}[\boldsymbol{x}(t), \boldsymbol{u}(t), t] \tag{27}$$

$$\text{边值条件：} \quad \boldsymbol{x}(t_0) = \boldsymbol{x}_0\,,\ \boldsymbol{x}(t_\perp) = \boldsymbol{x}_\perp \tag{28}$$

$$\text{目標函数：} \quad J = \int_{t_0}^{t_\perp} L[\boldsymbol{x}(t), \boldsymbol{u}(t), t]dt \tag{29}$$

要找的是最优控制 $\boldsymbol{u}^*(t)$。

Lagrange multiplier 是找极大/小值的常用方法：如果我们要找：

$$\max\ f(x) \quad \text{subject to} \quad g(x) = 0 \tag{30}$$

Lagrange 建议我们建构 Lagrangian 函数：

$$L(x, \lambda) = f(x) - \lambda g(x) \tag{31}$$

然后求解：

$$\nabla_{x,\lambda} L = 0 \tag{32}$$

现在将 Lagrange multiplier 方法应用到我们的问题上，会发现新的目标函数是：

$$J = \int_{t_0}^{t_\perp} \{L + \boldsymbol{\lambda}^T(t)\,[f(\boldsymbol{x}, \boldsymbol{u}, t) - \dot{\boldsymbol{x}}]\}dt \tag{33}$$

因此可以引入一个新的标量函数 H，即 Hamiltonian：

$$H(\boldsymbol{x}, \boldsymbol{u}, t) = L(\boldsymbol{x}, \boldsymbol{u}, t) + \boldsymbol{\lambda}^T(t) f(\boldsymbol{x}, \boldsymbol{u}, t) \tag{34}$$

物理学上，$\boldsymbol{f}$ 的单位是速度，而 L 的单位是能量，所以 λ 应该具有 动量 的单位。

极小值原理

Lev Pontryagin (1908-1988) 提出了 极小值原理，是经典变分法的推广。经典变分法的最优条件是：

$$\frac{\partial H}{\partial \boldsymbol{u}} = \boldsymbol{0} \tag{35}$$

极小值原理将最优条件改成是：

$$\min_{u\in\Omega} H(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{u}, t) = H(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{u}^*, t) \tag{36}$$

即是说：在最优轨迹 $\boldsymbol{x}^*(t)$ 和最优控制 $\boldsymbol{u}^*(t)$ 上，H 取最小值。它的好处是，当 $\dfrac{\partial H}{\partial \boldsymbol{u}}$ 不连续或不存在时，或者 $\boldsymbol{u}$ 受其他约束时，也可以应用。

粗略来说，极小值原理比经典变分法更一般，而动态学习又比极小值原理更一般。

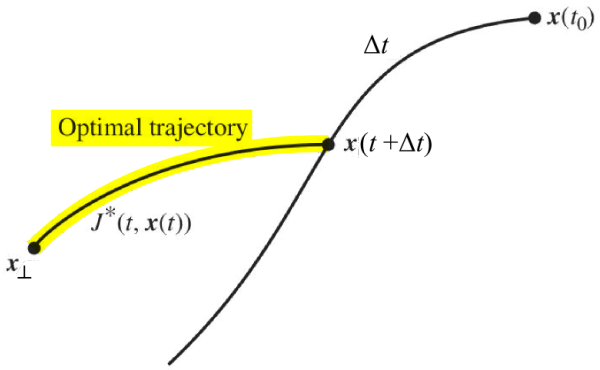
Hamilton-Jacobi-Bellman 方程

- Stanislaw Zak (2003): *Systems and control*

用动态规划的 **Bellman optimality condition** 可以推导出微分形式的 Hamilton-Jacobi-Bellman 方程。重温一下，Bellman 最优条件说的是：「从最优路径末端切去一小截之后，馀下的还是最优路径。」它通常写成如下的 recursive 形式：

$$\boxed{\text{最优路径}} = \text{在小段上选取最大奖励} + \boxed{\text{馀下的最优路径}} \tag{37}$$

$$J_t^* = \max_u \{ \boxed{\text{奖励(u, t)}} + J_{t-1}^* \} \tag{38}$$



(39)

我们在时间 interval 的开端切出一小段：

$$[t_0, t_\perp] = [t_0, t_0 + \Delta t] \cup [t_0 + \Delta t, t_\perp] \tag{40}$$

我们想优化的目标函数是：

$$J^*(t,\boldsymbol{x},\boldsymbol{u}) = \min_u\{\int_{t_0}^{t+\Delta t} L d\tau + \int_{t+\Delta t}^{t_\perp} L d\tau + \Phi(t_\perp, \boldsymbol{x}_\perp)\} \tag{41}$$

根据 Bellman 条件，目标函数变成：

$$J^*(t,\boldsymbol{x}) = \min_u\{\int_{t_0}^{t+\Delta t} L d\tau + J^*(t+\Delta t, \boldsymbol{x}(t+\Delta t))\} \tag{42}$$

用 Taylor series 展开右面的 J^*：

$$J^* + \frac{\partial J^*}{\partial t}\Delta t + \frac{\partial J^*}{\partial \boldsymbol{x}}(\boldsymbol{x}(t+\Delta t) - \boldsymbol{x}(t)) + \text{H.O.T.} \tag{43}$$

左右两边的 J^* 互相消去，而且 $\boldsymbol{x}(t+\Delta t) - \boldsymbol{x}(t) \approx \dot{\boldsymbol{x}}\Delta t$，於是有：

$$0 = \min_u\{\int_{t_0}^{t+\Delta t} L d\tau + \frac{\partial J^*}{\partial t}\Delta t + \frac{\partial J^*}{\partial \boldsymbol{x}}\dot{\boldsymbol{x}}\Delta t + \text{H.O.T.}\} \tag{44}$$

又由於 Δt 很小，而且 $\dot{\boldsymbol{x}} = \boldsymbol{f}$，所以：

$$0 = \min_u\{L\Delta t + \frac{\partial J^*}{\partial t}\Delta t + \frac{\partial J^*}{\partial \boldsymbol{x}}\boldsymbol{f}\Delta t + \text{H.O.T.}\} \tag{45}$$

全式除以 Δt 并令 $\Delta t \to 0$：

$$0 = \frac{\partial J^*}{\partial t} + \min_u\{L + \frac{\partial J^*}{\partial \boldsymbol{x}}\boldsymbol{f}\} \tag{46}$$

记得 Hamiltonian 的定义是 $H = L + \frac{\partial J^*}{\partial \boldsymbol{x}}\boldsymbol{f}$，所以得到想要的结果：

$$\boxed{\text{Hamilton-Jacobi equation}} \quad 0 = \frac{\partial J^*}{\partial t} + \min_u H \tag{47}$$

这个方程和量子力学中的 **Schrödinger equation** 很相似：

$$i\hbar\frac{\partial}{\partial t}\Psi(x,t) = \left[V(x,t) + \frac{-\hbar^2}{2\mu}\nabla^2\right]\Psi(x,t). \tag{48}$$

其中 Ψ 类似於我们的 J （或许 Ψ 是自然界希望取极值的某种东西?）

3.4 Symplectic 结构

- Stephanie Singer (2001): *Symmetry in mechanics – a gentle, modern introduction*
- 锺万勰 2011：《力、功、能量与辛数学》

Symplectic 的拉丁文意思是「互相交错 (intertwined)」，它用来描述 Hamiltonian 系统的几何结构。中文译作「辛」是音译。Symplectic 概念是 Hermann Weyl 研究 Hamilton 系统的对称性时在 1939 年提出的。

在数值计算上，处理 Hamilton 系统时，如果算法尊重 symplectic 结构（叫 symplectic integrators），会比一般的算法更准确；而一般解微分方程的算法，例如 Euler 算法和 Runge-Kutta 算法，有时会给出错误的结果。

举例来说，从 Hamiltonian 的角度来看，动量 p (momentum) 和速度 v (velocity) 是成对偶的，p 总是伴随 v 出现，因为 $p \cdot v = mv^2$ 的单位是能量。

举另一个例子，假设我们有两个用来定义系统状态的向量：

$$x_1 = \begin{pmatrix} s_1 \\ f_1 \end{pmatrix}, \quad x_2 = \begin{pmatrix} s_2 \\ f_2 \end{pmatrix} \tag{49}$$

其中 s 是位移（单位是长度），f 是力。这两个向量的「辛内积」定义为：

$$\begin{aligned} \langle x_1, x_2 \rangle &= x_1^T \, J \, x_2 \\ &= \begin{pmatrix} s_1 \\ f_1 \end{pmatrix}^T \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} \begin{pmatrix} s_2 \\ f_2 \end{pmatrix} \\ &= f_2 s_1 - f_1 s_2 \end{aligned} \tag{50}$$

其中矩阵 J 就是辛的微分形式 ω 的结构矩阵（下述）。由於 $f \cdot s$ 表示的是「所做的功」，上式表示的是

(状态 1 的力对状态 2 的位移所做的功)−
(状态 2 的力对状态 1 的位移所做的功) (51)

也就是「相互功」，辛正交则 $\langle x_1, x_2 \rangle = 0$，代表 work reciprocity（功的互等），所以辛几何是一种关於能量的代数。

在微分几何里，研究抽象的 Hamiltonian systems，会发现 symplectic 结构。这结构用微分流形 M 及其上的一个微分形式 (differential form) ω 来定义。需要一些微分几何的基础.....

Vectors and co-vectors

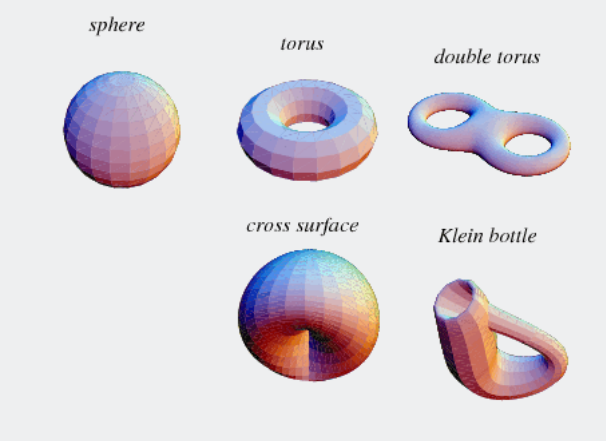
Vector 和 co-vector 之间的关系，可以看成是「d 别人的东西」和「被别人 d 的东西」，这里 d 表示微分。

「d 别人的东西」是线性的 differential operators，记作 $\frac{\partial}{\partial x_1}$、$\frac{\partial}{\partial x_2}$ 等；它们很自然地组成一个 vector space $T_x M$。

「被别人 d 的东西」是一些线性的微分形式，记作 dx_1、dx_2 等；它们属於 T_xM 的 dual space。

Manifolds

基本上「流形」的意思是「弯曲的空间」，它们局部地近似於 Euclidean 空间 $\mathbb{R}^n$，局部的座标可以分段用一组微分映射来描述，这些 maps 叫 charts。



(52)

Phase space

「相位空间」指的是力学系统里，i 个粒子的位置 x_i 和动量 p_i 合并而成的 $(\boldsymbol{x}, \boldsymbol{p})$ 空间。但 configuration space 指的是所有可容许的位置 $\boldsymbol{x}$ 的空间。

Vector fields, differential forms, Hamiltonian flow

根据 Hamilton 方程，再用微分的 chain rule 可以得到：

$$\frac{d}{dt} = \frac{\partial H}{\partial \boldsymbol{p}} \frac{\partial}{\partial \boldsymbol{x}} - \frac{\partial H}{\partial \boldsymbol{x}} \frac{\partial}{\partial \boldsymbol{p}} \tag{53}$$

它是一个微分算子，亦即是向量场；它有个特别的名字叫 Hamiltonian flow $\vec{H}$（很多书记作 X_H）：

$$\vec{H} := \frac{d}{dt} = \{\cdot, H\} \tag{54}$$

可以看出 $\vec{H}H = \{H, H\} = \frac{dH}{dt} = 0$ 就是能量守恒的形式。

Hamilton 系统的动态方程就是：

$$\dot{\boldsymbol{x}} = \vec{H} \tag{55}$$

所以在我们的智能系统中，RNN 可以看成是 $\vec{H}$。

$$\omega = \sum_i dx_i \wedge dp_i \tag{56}$$

$$\omega(\vec{H}, \cdot) = dH \tag{57}$$

我暂时不很明白它的意义。

我们说 Hamiltonian flow 保持 (preserve) 辛结构。假设 $\Gamma_t(\boldsymbol{x}_0) = \boldsymbol{x}(t)$ 描述 Hamiltonian flow 的轨迹；$\Gamma_0(\boldsymbol{x}) \equiv \boldsymbol{x}$。The **pullback** of ω along Γ is still ω:

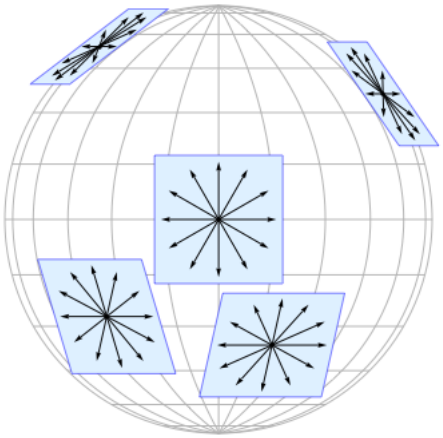
$$\Gamma_t^* \omega = \omega \tag{58}$$

$$\vec{H} H = 0 \quad \text{is equivalent to} \quad \Gamma_t^* \omega = \omega \tag{59}$$

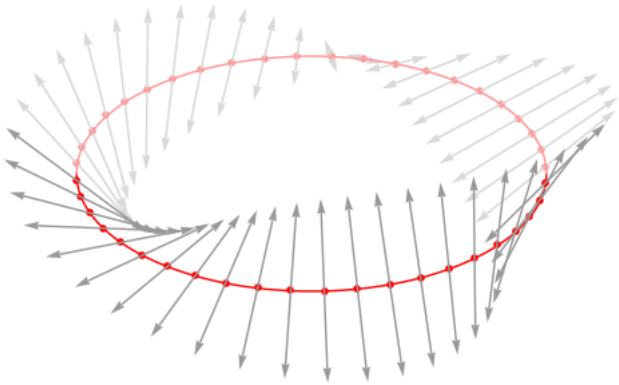
Tangent and co-tangent bundle

在流形上每点有一个 tangent space，所谓 tangent bundle 是指流形上每点 x 的 tangent space $T_x M$ 的总和：

$$TM := \bigcup_{x \in M} T_x M \tag{60}$$



Tangent bundle on a 2-sphere (61)



Bundle with Möbius strip topology

(62)

粗略地，tangent bundle 可以看成是 $M \times T_x M$，而 M 和 $T_x M$ 的维数都是 n，所以 tangent bundle 的维数是 $2n$。在力学上，**cotangent bundle** 就是相位空间 $(\boldsymbol{x}.\boldsymbol{p})$ 的空间 (The phase space of a mechanical system is the cotangent bundle of its configuration space)。

Push forward, pull back

Energy conservation, area form

(Stephanie §4.4)

Symmetry of Hamiltonian system, Lie groups

Symplectic groups 是一些保存辛结构的变换 T 的群：

$$\omega(Tx, Ty) = \omega(x, y) \quad \forall x, y \in V \tag{63}$$

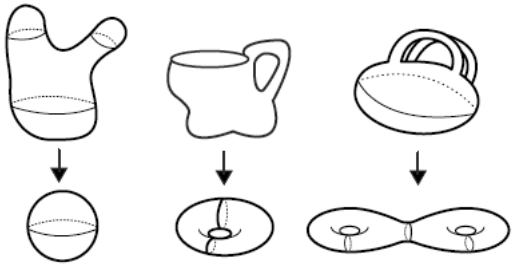
V 是向量空间。在 V 上的 symplectic 变换的全体记作 $Sp(V)$。

Momentum maps

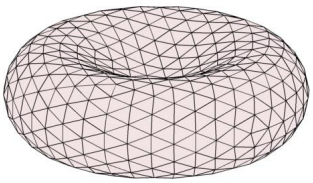
3.5 动态系统理论

Floer homology

Homology （同调论） 研究的是空间中「有没有穿洞」的樸拓结构。最简单的 **singular homology** 是将空间用三角形剖分 (triangulation)，然后透过著名的 Euler formula $V + F = E + 2$ 及其扩充，让我们可以计算 Euler characteristic χ，此即空间穿洞的个数。

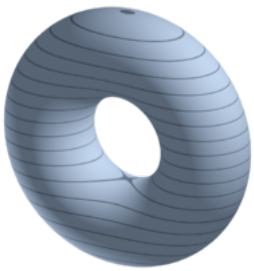


(64)



(65)

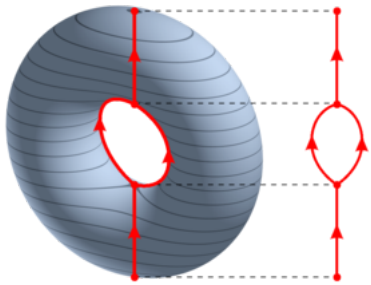
当这些三角形剖分趋於无穷小时，我们得到用微分形式 (differential forms) 描述的 homology，即 **de Rham homology**。



(66)

在流形上定义一个 potential function，例如简单的 height function，就可以做 Morse theory，这时每个点可以根据势能函数向下流 (gradient flow)，流到一些最低位置，它们是临界点 (critical points)。Morse decomposition 将空间用这些临界点分割（流到同一临界点的 flows 认作同一 equivalent class），得到的是

Morse homology。



(67)

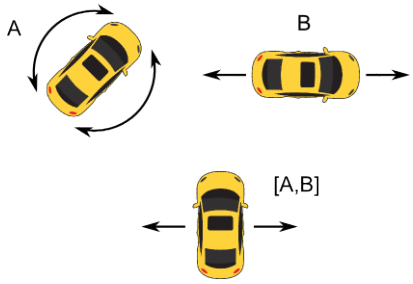
Floer homology 是 Morse homology 的**无限维空间**版本，比较难计算。

Conley theory

Entropy, ergodicity

Lie algebra 的另一应用

例如一架车可以有两个基本动作：(A) 绕中心旋转、或 (B) 前后行驶，它们的 Lie 括号 $[A, B]$，产生出的动作是左右方向行驶，这类似於「平行泊车」的时候，车子的移动方向：



(68)

（可控性与 “reachable” 概念。）

Stable, unstable, and center manifold

Smale horseshoe

4　最优控制的计算方法

直接法

间接法

Lyapunov 函数第一方法

Lyapunov 函数第二方法

5 Jacobian 神经网络学习算法

经典的神经网络 Back Prop 学习算法，它是一个 error-driven 算法，但在很多人工智能的实际应用中，不存在唯一的「理想答案」，而是根据正或负的奖励 (reward) 学习。当答案正确时，奖励 $> 0, \text{error} = 0$；当答案不正确时，奖励 < 0，但 error 仍是不知道的（因为不知道理想答案）。简言之，就是不能用 error-driven 学习。

所以我想出了一个 reward-driven 的学习法：假设神经网络将 $\boldsymbol{x}_0 \mapsto \boldsymbol{y}_0$，它通常也会将 $\boldsymbol{x}_0$ 的邻域 map 到 $\boldsymbol{y}_0$ 的邻域。如果我们想「加强」这个映射，可以将「更大的 $\boldsymbol{x}_0$ 的邻域」映射到「接近 $\boldsymbol{y}_0$ 的邻域」。

这种算法对人工智能应该很重要，暂时我还想不出有什么其他办法，可以做到 [深度] 神经网络的 reward-driven 学习。

将这思想更准确化，可以将 feed-forward 神经网络的构造看成是这样的：

$$\boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x}) \tag{69}$$

$$\boldsymbol{y} = \sigma \overset{L}{W} \ldots \sigma \overset{\ell}{W} \ldots \sigma \overset{1}{W} \boldsymbol{x} \tag{70}$$

其中 W 代表每一层 (layer) ℓ 的矩阵。

F 的**反方向**是：

$$\boldsymbol{x} = \boldsymbol{F}^{-1}(\boldsymbol{y}) \tag{71}$$

$$\boldsymbol{x} = \overset{1}{\overleftarrow{W}} \bar{\sigma} \ldots \overset{\ell}{\overleftarrow{W}} \bar{\sigma} \ldots \overset{L}{\overleftarrow{W}} \bar{\sigma} \boldsymbol{y} \tag{72}$$

注意：$\overleftarrow{W} = W^{-1}$，$\bar{\sigma} = \sigma^{-1}$，形状不同。

假设在 $\boldsymbol{x}$ 空间有体积元 U，经过 $\boldsymbol{F}$ 变换成 $\boldsymbol{y}$ 空间的体积元 V，那么：

$$U = |J| \cdot V \tag{73}$$

$J = \left[\dfrac{\partial \boldsymbol{F}(x)}{\partial x}\right]$ 叫 Jacobian 矩阵。

在我们的情况下，$|J| = \left|\dfrac{\partial \boldsymbol{F}^{-1}(\boldsymbol{y})}{\partial \boldsymbol{y}}\right|$ 在 $\boldsymbol{y}_0$ 的值，代表「单位体积元由 $\boldsymbol{y}_0 \mapsto \boldsymbol{x}_0$ 的变化率」。（下面会看到，$\boldsymbol{F}$ 和 $\boldsymbol{F}^{-1}$ 的正/反方向不太重要，因为基本上不影响计算复杂度。）

每次得到**正奖励**，我们会令 Jacobian $|J|$ 增加一点：

$$|J| := \det \left[\frac{\partial \boldsymbol{F}^{-1}(\boldsymbol{y})}{\partial \boldsymbol{y}}\right]_{n \times n} \tag{74}$$

下标表示那是一个 $n \times n$ 矩阵。

其实 Jacobian 矩阵的意义就是：

$$J = \left[\frac{\partial \text{ 输出}}{\partial \text{ 输入}}\right] \tag{75}$$

神经网络的输入和输出都是 $\dim n$，所以 Jacobian 很自然是 $n \times n$ 矩阵。

用**梯度下降法**，我们需要计算这些梯度：$\left[\frac{\partial |J|}{\partial \boldsymbol{W}}\right]$，总数是网络中的 weights 的个数 $= \sum m_\ell$。

要用到 determinant 的微分公式：

$$\frac{d}{dt}|A(t)| = tr(\text{adj}(A) \cdot \frac{dA(t)}{dt}) \tag{76}$$

$$\text{adj}(A) := |A| \cdot A^{-1} \tag{77}$$

换句话说，对於每个权重 $w := \overset{\ell}{W}_{ij}$，我们要计算：

$$\frac{\partial}{\partial w}|J| = tr(|J| \cdot J^{-1} \cdot \left[\frac{\partial J}{\partial w}\right]) \tag{78}$$

注意：$|J|$ 和 J^{-1} 是 $\boldsymbol{y}_0$ 的函数，只需在大 loop 外一次过计算。

问题是，计算 $\left[\frac{\partial J}{\partial w}\right]_{n\times n}$ 的时候：

$$\frac{\partial J}{\partial w} = \frac{\partial}{\partial w}\frac{\partial \boldsymbol{F}^{-1}}{\partial y} = \frac{\partial}{\partial w}\frac{\partial}{\partial \boldsymbol{y}} \overset{1}{\mathcal{W}} \sigma \cdots \overset{\ell}{\mathcal{W}} \sigma \cdots \overset{L}{\mathcal{W}} \sigma \boldsymbol{y} \tag{79}$$

这牵涉到用 w 对 W^{-1} 的分量微分，可以想像就算计了出来也会是极复杂的。解决办法是，索性「**本末倒置**」，用 $\overset{\ell}{\mathcal{W}}$ 来定义神经网络，然后在 forward propagation 时才用 $W = \overset{\ell}{\mathcal{W}}^{-1}$ 计算。

J 的分量写出来是：

$$J_{ij} = \frac{\partial \boldsymbol{F}_i^{-1}}{\partial y_j} = \frac{\partial}{\partial y_j}\left[\overset{1}{\mathcal{W}} \sigma \cdots \overset{\ell}{\mathcal{W}} \sigma \cdots \overset{L}{\mathcal{W}} \sigma \boldsymbol{y}\right]_i =: \nabla^1_{ij} \tag{80}$$

$$\begin{cases} \nabla^1_{ij} & := \sum_{k_1}[\overset{1}{\mathcal{W}}_{ik_1} \sigma'(y^2_j)\nabla^2_{ij}] \\ \nabla^\ell_{ij} & := \sum_{k_\ell}[\overset{\ell}{\mathcal{W}}_{k_{\ell-1}k_\ell} \sigma'(y^{\ell+1}_j)\nabla^{\ell+1}_{ij}] \\ \nabla^L_{ij} & := \overset{L}{\mathcal{W}}_{k_{L-1}j} \sigma'(y_j) \end{cases} \tag{81}$$

这情况完全类似於经典 Back Prop，以上只是 chain rule 的应用，∇^ℓ 将每层用 chain rule 分拆开来，所以 ∇ 又叫 “local gradient”。上式就是整个网络的**反向传递**，其中每个 weight 出现 exactly 一次。

但工作还未完，我们要计算 $\frac{\partial J_{ij}}{\partial \wr} = \dot{\nabla}^1_{ij}$。（定义 $\wr := \overset{\ell}{\mathbb{W}}_{gh}$，$k_0 := i$，$k_L := j$）

注意：$\boldsymbol{x} = \boldsymbol{F}^{-1}(\boldsymbol{y})$，所以 $\boldsymbol{y}$ 是**自变量**，$\wr$ 不影响 $\boldsymbol{y}$，所以 $\frac{\partial \boldsymbol{y}}{\partial \wr} \equiv 0$。

$\wr$ 必会是 $\mathbb{W}$ 的其中一元，但如果 $\wr \notin \mathbb{W}$，以下的项微分后都会变成 0：

$$\begin{cases} \dot{\nabla}^1_{ij} = \sum_{k_1} \left[\overset{1}{\mathbb{W}}_{ik_1} \ominus'(y^2_j) \dot{\nabla}^2_{ij} \right] \\ \dot{\nabla}^\ell_{ij} = \sum_{k_\ell} \left[\overset{\ell}{\mathbb{W}}_{k_{\ell-1}k_\ell} \ominus'(y^{\ell+1}_j) \dot{\nabla}^{\ell+1}_{ij} \right] \\ \dot{\nabla}^L_{ij} = \quad \overset{L}{\mathbb{W}}_{k_{L-1}j} \ominus'(y_j) \equiv 0 \end{cases} \tag{82}$$

所以实际上只剩下一项：

$$\frac{\partial J_{ij}}{\partial \wr} = \sum_{k_1} \left[\overset{1}{\mathbb{W}}_{k_0 k_1} \ominus'(y^2_j) \dots \sum_{k_\ell} \left[\overset{\ell}{\mathbb{W}}_{k_{\ell-1}k_\ell} \ominus'(y^{\ell+1}_j) \dots \begin{cases} \dots \ominus'(y^{\ell+1}_j) \nabla^{\ell+1}_{ij} \big] \big] \\ \dots \ominus'(y^{\ell+1}_j) \big] \big] \end{cases} \quad \text{if } \wr \in \text{last layer} \right. \tag{83}$$

上式的意思是：每层 layer 重复一块 $\left[\sum \mathbb{W} \ominus\right]$，直到遇到 $\wr = \overset{\ell}{\mathbb{W}}_{gh}$，则用结尾形式取代之。

和经典 Back Prop 不同的是，上式只是 $n \times n$ 矩阵中的一个元素，从复杂度而论，每个 weight 的 ∇ 计算，增加了起码 n^2 倍的复杂度（虽然其计算上可以共用一些结果）。记住 $n = \dim \boxed{\text{状态空间}}$。

可以这样理解：每个 weight 的调教，需要计算这个 weight 对 Jacobian 的影响，而那 Jacobian 是**整个**网络的特性。关键似乎就在於每个 weight 对 Jacobian 的**影响**。

现在回看更高层次的这个式子：

$$\frac{\partial}{\partial w}|J| = tr(|J| \cdot J^{-1} \cdot \left[\frac{\partial J}{\partial w}\right]) \tag{84}$$

$$= |J| \cdot tr(\left[\frac{\partial y}{\partial x}\right] \cdot \left[\frac{\partial}{\partial w}\frac{\partial x}{\partial y}\right]) \tag{85}$$

$$= |J| \cdot \sum_{ij} \left(\frac{\partial y}{\partial x}\right)_{ij} \left(\frac{\partial}{\partial w}\frac{\partial x}{\partial y}\right)_{ij} \tag{86}$$

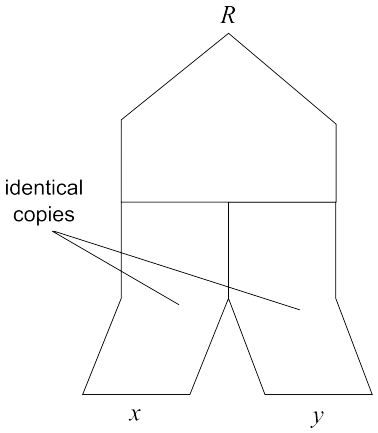
上式中最重要（最慢）的是那 $(i,j) \in n \times n$ 求和。裡面的第一個因子是 Jacobian J，第二个因子是我们刚计算了的 $\nabla_w J^{-1}$。

Back Prop 的 ∇ 形式上是 $\frac{\partial \text{输出}}{\partial \text{w}}$，我们的 ∇ 形式是 $\left[\frac{\partial}{\partial w}\frac{\partial \text{输入}}{\partial \text{输出}}\right]_{n\times n}$。

其实我们只需要计算 $\nabla_w |J|$ 的**大约方向**。暂时我在代码中的做法是：忽略式 (86) 中较小的项，那就不需做足 n^2 个乘积。

或者可不可以将 $|J|(\geqslant)$ 看成是一个 weight $\geqslant$ 的函数，然后用它的 Taylor series expansion 来近似？

Notes



(87)

或者，照旧是 feedforward network，但 somehow 它的学习是基於某 reward function。问题是这 reward function 从何而来？R 可以是一个全部 W 的函数，是由我们任意定义的。

6 Jacobian learning algorithm

The classical Back Prop algorithm is **error-driven**, but in many AI problems the "correct" answers are not given, instead the feedback is provided via **rewards**. When an answer is correct, the reward $R > 0$, error $\mathscr{E} = 0$; when the answer is incorrect, $R < 0$, but $\mathscr{E}$ is still unknown (because we don't know the correct answer). In other words, error-driven learning is inapplicable.

So I thought of a reward-driven learning method: assume the neural network maps $\boldsymbol{x}_0 \mapsto \boldsymbol{y}_0$, usually it also maps the neighborhood of $\boldsymbol{x}_0$ to the neighborhood of $\boldsymbol{y}_0$. If we wish to "strengthen" this pair of mapping, we can make a **bigger** neighborhood of $\boldsymbol{x}_0$ map to the same neighborhood close to $\boldsymbol{y}_0$. We will make this precise.

This type of learning algorithm should be very useful to AI, and currently the author is not aware of other alternatives for training [deep] neural networks via rewards.

A feed-forward neural network can be constructed this way:

$$\boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x}) \tag{88}$$

$$\boldsymbol{y} = \sigma \overset{L}{W} \ldots \sigma \overset{\ell}{W} \ldots \sigma \overset{1}{W} \boldsymbol{x} \tag{89}$$

where W represents the matrix of **weights** on each layer ℓ.

The **inverse** of F is:

$$\boldsymbol{x} = \boldsymbol{F}^{-1}(\boldsymbol{y}) \tag{90}$$

$$\boldsymbol{x} = \overset{1}{\overleftarrow{W}} \bar{\sigma} \ldots \overset{\ell}{\overleftarrow{W}} \bar{\sigma} \ldots \overset{L}{\overleftarrow{W}} \bar{\sigma} \boldsymbol{y} \tag{91}$$

Note: $\overleftarrow{W} = W^{-1}$, $\bar{\sigma} = \sigma^{-1}$, the shape is different.

Assume that in the space of $\boldsymbol{x}$ there is a volume element U, which transforms via $\boldsymbol{F}$ to a volume element V in the space of $\boldsymbol{y}$, then:

$$U = |J| \cdot V \tag{92}$$

$J = \left[\dfrac{\partial \boldsymbol{F}(x)}{\partial x} \right]$ is called the **Jacobian** matrix.

In our case, the value of $|J| = \left| \dfrac{\partial \boldsymbol{F}^{-1}(\boldsymbol{y})}{\partial \boldsymbol{y}} \right|$ at $\boldsymbol{y}_0$ represents "the change in volume from $\boldsymbol{y}_0 \mapsto \boldsymbol{x}_0$". (Below, we will see that the direction of $\boldsymbol{F}$ or $\boldsymbol{F}^{-1}$ is not too important, as either way the computational complexity is essentially the same.)

Every time we get a **positive reward**, we can let the Jacobian $|J|$ increase slightly:

$$|J| := \det \left[\frac{\partial \boldsymbol{F}^{-1}(\boldsymbol{y})}{\partial \boldsymbol{y}} \right]_{n \times n} \tag{93}$$

The subscript indicates that it is a $n \times n$ matrix.

The meaning of the Jacobian matrix is:

$$J = \left[\frac{\partial \text{ input}}{\partial \text{ output}} \right] \tag{94}$$

The neural network's input and output are both of $\dim n$, so the Jacobian is naturally an $n \times n$ matrix.

To use **gradient descent**, we need to calculate these gradients: $\left[\frac{\partial |J|}{\partial \boldsymbol{W}} \right]$, their total number is the number of weights in the network $= \sum \ell \, \#(\overset{\ell}{W})$.

We need the formula for the derivative of the determinant:

$$\frac{d}{dt}|A(t)| = tr(\text{adj}(A) \cdot \frac{dA(t)}{dt}) \tag{95}$$

$$\text{adj}(A) := |A| \cdot A^{-1} \tag{96}$$

In other words, for each weight $w := \overset{\ell}{W}_{ij}$, we need to calculate:

$$\frac{\partial}{\partial w}|J| = tr(|J| \cdot J^{-1} \cdot \left[\frac{\partial J}{\partial w} \right]) \tag{97}$$

Note: $|J|$ and J^{-1} are functions of $\boldsymbol{y}_0$, we only need to calculate them once outside the big loop.

Now a problem arises in the calculation of $\left[\frac{\partial J}{\partial w} \right]_{n \times n}$:

$$\frac{\partial J}{\partial w} = \frac{\partial}{\partial w} \frac{\partial \boldsymbol{F}^{-1}}{\partial y} = \frac{\partial}{\partial w} \frac{\partial}{\partial \boldsymbol{y}} \quad \overset{1}{\leftarrow} \bigcirc \cdots \overset{\ell}{\leftarrow} \bigcirc \cdots \overset{L}{\leftarrow} \bigcirc \, \boldsymbol{y} \tag{98}$$

This requires us to differentiation W^{-1} w.r.t. w; we can imagine the result would be very complicated. So we use a trick, by "reversing" the network, we use $\overleftarrow{W}$ to define the network weights, and then use $W = \overleftarrow{W}^{-1}$ during forward propagation.

The components of J are:

$$J_{ij} = \frac{\partial \boldsymbol{F}_i^{-1}}{\partial y_j} = \frac{\partial}{\partial y_j} \left[\overset{1}{\leftarrow} \bigcirc \cdots \overset{\ell}{\leftarrow} \bigcirc \cdots \overset{L}{\leftarrow} \bigcirc \, \boldsymbol{y} \right]_i =: \nabla^1_{ij} \tag{99}$$

$$\begin{cases} \nabla^1_{ij} & := \sum_{k_1} [\, \overset{1}{\mathbb{W}}_{ik_1} \sigma'(y^2_j) \nabla^2_{ij} \,] \\ \nabla^\ell_{ij} & := \sum_{k_\ell} [\, \overset{\ell}{\mathbb{W}}_{k_{\ell-1}k_\ell} \sigma'(y^{\ell+1}_j) \nabla^{\ell+1}_{ij} \,] \\ \nabla^L_{ij} & := \quad \overset{L}{\mathbb{W}}_{k_{L-1}j} \sigma'(y_j) \end{cases} \tag{100}$$

This situation is exactly analogous to the classical Back Prop algorithm; The above is just the application of the **chain rule**, with ∇^ℓ written separately for each layer, therefore ∇ is called the "local gradient". The above formula amounts to propagating the entire network one time, where every weight appears **exactly once**.

But our work is not finished yet; We need to calculate $\dfrac{\partial J_{ij}}{\partial \mathbb{w}} =: \dot{\nabla}^1_{ij}$.

(Let's define $\mathbb{w} := \overset{\ell}{\mathbb{W}}_{gh}$, $k_0 := i$, $k_L := j$)
$\mathbb{w}$ would be an element of $\overset{\ell}{\mathbb{W}}$, but if $\mathbb{w} \notin \overset{\ell}{\mathbb{W}}$, all the terms below would vanish:

$$\begin{cases} \dot{\nabla}^1_{ij} = \sum_{k_1} \left[\overset{1}{\mathbb{W}}_{ik_1} \sigma'(y^2_j) \dot{\nabla}^2_{ij} \right] \\ \dot{\nabla}^\ell_{ij} = \sum_{k_\ell} \left[\overset{\ell}{\mathbb{W}}_{k_{\ell-1}k_\ell} \sigma'(y^{\ell+1}_j) \dot{\nabla}^{\ell+1}_{ij} \right] \\ \dot{\nabla}^L_{ij} = \quad \overset{L}{\mathbb{W}}_{k_{L-1}j} \sigma'(y_j) \equiv 0 \end{cases} \tag{101}$$

So what is left over is just this term:

$$\frac{\partial J_{ij}}{\partial \mathbb{w}} = \sum_{k_1} \left[\overset{1}{\mathbb{W}}_{k_0 k_1} \sigma'(y^2_j) \cdots \sum_{k_\ell} \left[\overset{\ell}{\mathbb{W}}_{k_{\ell-1}k_\ell} \sigma'(y^{\ell+1}_j) \cdots \begin{cases} \cdots \sigma'(y^{\ell+1}_j) \nabla^{\ell+1}_{ij} \big] \big] \\ \cdots \sigma'(y^{\ell+1}_j) \big] \big] \end{cases} \right. \right. \quad \text{if } \mathbb{w} \in \text{last layer} \tag{102}$$

The above formula means: For each layer we repeat a block of $\left[\sum \mathbb{w} \sigma\right]$, until we encounter $\mathbb{w} = \overset{\ell}{\mathbb{W}}_{gh}$, then we replace with the terminal form.

In contrast to classical Back Prop, the above formula gives us only one element in an $n \times n$ matrix; From the complexity point of view, calculating the ∇ for each weight is at least n^2 times as costly as classical Back Prop (even though we may re-use some intermediate computation results). Recall that $n = \dim \boxed{\text{state space}}$.

We can understand it thusly: For each weight we try to calculate its influence towards the Jacobian, but the Jacobian is a **global** property of the network. The key seems to lie in how each weight **influences** the Jacobian.

Now let's look back at this higher-level formula:

$$\frac{\partial}{\partial w}|J| = tr(|J| \cdot J^{-1} \cdot \left[\frac{\partial J}{\partial w}\right]) \tag{103}$$

$$= |J| \cdot tr(\left[\frac{\partial y}{\partial x}\right] \cdot \left[\frac{\partial}{\partial w}\frac{\partial x}{\partial y}\right]) \tag{104}$$

$$= |J| \cdot \sum_{ij} \left(\frac{\partial y}{\partial x}\right)_{ij} \left(\frac{\partial}{\partial w}\frac{\partial x}{\partial y}\right)_{ij} \tag{105}$$

The most critical (slowest) part is the $(i, j) \in n \times n$ summation. The first factor inside $\sum$ is the Jacobian J, the second factor is the $\nabla_w J^{-1}$ that we just calculated.

Back Prop's ∇ has the form $\dfrac{\partial \boxed{\text{output}}}{\partial \boxed{\text{weights}}}$

whereas our ∇ has the form $\left[\dfrac{\partial}{\partial \boxed{\text{weights}}} \dfrac{\partial \boxed{\text{input}}}{\partial \boxed{\text{output}}}\right]_{n \times n}$.

In fact we just need to calculate the **approximate** direction and size of $\nabla_w |J|$. Currently in our code we use this trick: ignore the smaller terms in (105), so we don't need to do all of n^2 products.

Or perhaps we can regard $|J|(\geq)$ as a function of the weight $\geq$, and then use its Taylor series expansion to approximate?

References

1. Lo. Dynamical system identification by recurrent multilayer perceptrons. *Proceedings of the 1993 World Congress on Neural Networks*, 1993.
2. Mikolov, Sutskever, Chen, Corrado, and Dean. Efficient estimation of word representations in vector space. *Proceedings of workshop at ICLR*, 2013.
3. Siegelmann and Sontag. Turing computability with neural nets. *Applied Mathematics Letters, vol 4, p77-80*, 1991.